Group Ratio Round-Robin: O(1) Proportional Share Scheduling for Uniprocessor and Multiprocessor Systems

Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein[*]*, and Haoqiang Zheng*
Department of Computer Science
Columbia University
Technical Report CUCS-028-04
July 2004

Abstract

Proportional share resource management provides a flexible and useful abstraction for multiplexing time-shared resources. We present Group Ratio Round-Robin (GR^3), the first proportional share scheduler that combines accurate proportional fairness scheduling behavior with $O(1)$ scheduling overhead on both uniprocessor and multiprocessor systems. GR^3 uses a novel client grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this grouping strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm. GR^3 can provide fairness within a constant factor of the ideal generalized processor sharing model for client weights with a fixed upper bound and preserves its fairness properties on multiprocessor systems. We have implemented GR^3 in Linux and measured its performance against other schedulers commonly used in research and practice, including the standard Linux scheduler, Weighted Fair Queueing, Virtual-Time Round-Robin, and Smoothed Round-Robin. Our experimental results demonstrate that GR^3 can provide much lower scheduling overhead and much better scheduling accuracy in practice than these other approaches.

1 Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing processor resources among a set of clients with associated weights. However, developing processor scheduling mechanisms that combine good proportional fairness scheduling behavior with low scheduling overhead has been difficult to achieve in practice. For many proportional share scheduling mechanisms, the time to select a client for execution grows with the number of clients. For server systems which may service large numbers of clients, the scheduling overhead of algorithms whose complexity grows linearly with the number of clients can waste more than 20 percent of system resources [2] for large numbers of clients. Furthermore, little work has been done to provide proportional share scheduling on small-scale multiprocessor systems, which are increasingly common. Over the years, a number of scheduling mechanisms have been proposed, and a significant amount of progress has been made. However, previous mechanisms have either superconstant overhead, or less-than-ideal fairness properties.

We introduce Group Ratio Round-Robin (GR^3), a proportional share scheduler that provides constant fairness bounds on proportional sharing accuracy with $O(1)$ scheduling overhead for both uniprocessor and small-scale multiprocessor systems. In designing GR^3, we observed that accurate, low-overhead proportional sharing is easy to achieve when scheduling a set of clients with equal processor allocations, but is harder to do when clients require very different processor allocations. Based on this observation, GR^3 uses a novel client grouping strategy to organize clients into groups of similar processor allocations which can be more easily scheduled. Using this grouping strategy, GR^3 combines the benefits of low overhead round-robin execution with a novel ratio-based scheduling algorithm. Specifically, we show that with only $O(1)$ overhead, GR^3 provides fairness within $O(g^2)$ of the ideal Generalized Processing Sharing (GPS) model [13], where g, the number of groups, is in practice a small constant that grows at worst logarithmically with the largest client weight. Moreover, we show how GR^3MP, an extension of GR^3, can be successfully applied to multiprocessor systems, preserving its worst-case time complexity and fairness properties.

GR^3 is simple to implement and can be easily incorporated into existing scheduling frameworks in commercial operating systems. We have implemented a prototype GR^3 processor scheduler in Linux, and compared our GR^3 Linux prototype against schedulers commonly used in practice and research, including the standard Linux scheduler, Weighted Fair Queueing [6], Virtual-Time Round-Robin [14], and Smoothed Round-Robin [4]. We have conducted extensive simulation studies and kernel measurements on micro-benchmarks and real applications. Our results show that GR^3 can provide more than an order of magnitude better proportional sharing accuracy than these other schedulers. Furthermore, our results show that GR^3 achieves this accuracy with lower scheduling overhead – more than an order of magnitude less than the standard Linux scheduler and typical Weighted Fair Queueing im-

[*]also in Department of IEOR, Columbia University.

plementations. For multiprocessors, we also provide better fairness with scheduling overhead that is an order of magnitude less than the standard Linux scheduler. These results demonstrate that GR^3 can in practice deliver better proportional share control with lower scheduling overhead than these other approaches.

This paper presents the design, analysis, and evaluation of GR^3. Sections 2 and 3 present the GR^3 scheduling algorithm for uniprocessor and multiprocessor systems. Section 4 analyzes the fairness and complexity of GR^3. Section 5 presents experimental results. Section 6 discusses related work.

2 GR^3 Scheduling

Proportional share scheduling has a clear colloquial meaning: given a set of clients with associated weights, a proportional share scheduler should allocate resources to each client in proportion to its respective weight. Without loss of generality, we can model the process of scheduling a time-multiplexed resource among a set of clients in two steps: 1) the scheduler orders the clients in a queue, 2) the scheduler runs the first client in the queue for its *time quantum*, which is the maximum time interval the client is allowed to run before another scheduling decision is made. We refer to the units of time quanta as time units (tu) in this paper rather than an absolute time measure such as seconds.

Based on the above scheduler model, a scheduler can achieve proportional sharing in one of two ways. One way, often called fair queueing [6, 15, 24, 9, 20, 5] is to adjust the frequency that a client is selected to run by adjusting the position of the client in the queue so that it ends up at the front of the queue more or less often. However, adjusting the position of the client in the queue typically requires sorting clients based on some metric of fairness, and has a time complexity that grows with the number of clients. The other way is to adjust the size of the time quantum of a client so that it runs longer for a given allocation. Weighted round-robin is the most common example of this approach. This approach is fast, providing constant time complexity scheduling overhead. However, allowing a client to monopolize the resource for a long period of time results in extended periods of unfairness to other clients which receive no service during those times.

GR^3 is a proportional share scheduler that matches the $O(1)$ time complexity of round-robin scheduling but provides much better proportional fairness guarantees in practice. At a high-level, the GR^3 scheduling algorithm can be briefly described in three parts:

1.**Client grouping strategy**: Clients are separated into groups of clients with similar weight values. Each group of order k is assigned clients with weights between 2^k to $2^{k+1} - 1$, where $k \geq 0$.

2.**Intergroup scheduling**: Groups are ordered in a list from largest to smallest group weight, where the group weight of a group is the sum of the weights of all clients in the group. Groups are selected in a round-robin manner based on the ratio of their group weights. If a group has already been selected more than its proportional share of the time, move on to the next group in the list. Otherwise, skip the remaining groups in the group list and start selecting groups from the beginning of the group list again. Since the groups with larger weights are placed first in the list, this allows them to get more service than the lower-weight groups at the end of the list.

3.**Intragroup scheduling**: From the selected group, a client is selected to run in a round-robin manner that accounts for its weight and previous execution history.

Using this client grouping strategy, GR^3 separates scheduling in such a way that reduces the need to schedule entities with skewed weight distributions. The client grouping strategy limits the number of groups that need to be scheduled since the number of groups grows at worst logarithmically with the largest client weight. Even a very large 32-bit client weight would limit the number of groups to no more than 32. The client grouping strategy also ensures that all clients within a group have weight within a factor of two. As a result, the intragroup scheduler never needs to schedule clients with skewed weight distributions. GR^3 groups are simple lists that do not need to be balanced; they do not require any use of more complex balanced tree structures.

2.1 GR^3 Definitions

We now define the state GR^3 associates with each client and group, and then describe in detail how GR^3 uses that state to schedule clients. Table 1 presents a list of terminology. In GR^3, a client has three values associated with its execution state: weight, deficit, and run state. Each client receives a resource allocation that is directly proportional to its *weight*. A client's *deficit* tracks the number of remaining time quanta the client has not received from previous allocations. A client's *run state* is an indication of whether or not the client can be executed. A client is *runnable* if it can be executed. For example for a CPU scheduler, a client would not be runnable if it is blocked waiting for I/O and cannot execute.

A group in GR^3 has a similar set of values associated with it: group weight, group order, group work, and current client. The *group weight* is defined as the sum of the corresponding weights of the clients in the group run queue. A group with *group order* k contains clients with weights between 2^k to $2^{k+1} - 1$. The *group work* is the total execution time clients in the group have received. The *current client* is the most recently scheduled client in the group's run queue.

In addition to the per client and per group state described, GR^3 maintains the following scheduler state: time quantum, group list, total weight, total work, and current

C_i	Client i. (also called 'task' i)
ϕ_i	The weight assigned to C_i.
D_{C_i}	The deficit of C_i.
N	The number of runnable clients.
Φ_T	The sum of the weights of all runnable clients: $\sum_{C_j} \phi_j$.
g	The number of groups.
$\|G\|$	The number of clients in group G.
G_i	i'th Group in the ordered list of groups.
$G(i)$	The group to which C_i belongs.
Φ_G	The group weight: $\sum_{C_i \in G} \phi_i$.
Φ_i	Shorthand notation for Φ_{G_i} .
σ_G	The order of group G.
W_C	The work of client C.
W_G	The group work of group G.
W_k	shorthand notation for W_{G_k}.
W_T	The sum of the group work of all groups.

Table 1: GR^3 Terminology

group. The *group list* is a sorted list of all groups containing runnable clients ordered from largest to smallest group weight, with ties broken by group order. The *total weight* is the sum of the weights of all runnable clients. The *total work* is the sum of the work of all groups. The *current group* is the most recently selected group in the group list.

2.2 Basic GR^3 Algorithm

We will initially only consider runnable clients in our discussion of the basic GR^3 scheduling algorithm. We will discuss dynamic changes in a client's run state in Section 2.3. We first focus on the development of the GR^3 intergroup scheduling algorithm and then discuss the development of the GR^3 intragroup scheduling algorithm.

The key idea behind the GR^3 **intergroup scheduling** algorithm is that we can choose the next group to schedule using only the state of successive groups in the group list. The basic idea is given a group G_i whose weight is x times larger than the group weight of the next group G_{i+1} in the group list, GR^3 will select group G_i x times for every time that it selects G_{i+1} in the group list to provide proportional share allocation among groups. To implement the algorithm, we maintain the the total work done by group i in a variable W_i. The algorithm then repeatedly executes the following simple routine:

Run a client from G_i; increment W_i
if
$$\frac{W_i + 1}{W_{i+1} + 1} > \frac{\Phi_i}{\Phi_{i+1}} \quad (1)$$
then increment i
else $i = 1$

The index i tracks the current group and is initialized to 1.

The intuition behind (1) is that we would like the ratio of the work of G_i and G_{i+1} to match the ratio of their respective group weights after GR^3 has finished selecting both groups. For each time a client from G_{i+1} is run, GR^3 would like to have run $\frac{\Phi_i}{\Phi_{i+1}}$ worth of clients from G_i. (1) says that GR^3 should not run a client from G_i and increment G_i's group work if it will make it impossible for G_{i+1} to catch up to its proportional share allocation by running one of its clients once.

To illustrate how the intergroup scheduling works, consider an example in which we have three clients C_1, C_2, and C_3, which have weights of 5, 2, and 1, respectively. The GR^3 grouping strategy would place each C_i in group G_i, ordering the groups by weight: G_1, G_2, and G_3 have orders 2, 1 and 0 and weights of 5, 2, and 1 respectively. GR^3 would start by selecting group G_1, running client C_1, and incrementing W_1. Based on (1), $\frac{W_1+1}{W_2+1} = 2 < \frac{\Phi_1}{\Phi_2} = 2.5$, so GR^3 would select G_1 again and run client C_1. After running C_1, G_1's work would be 2 so that the inequality in (1) would hold and GR^3 would then move on to the next group G_2 and run client C_2. Based on (1), $\frac{W_2+1}{W_3+1} = 2 \leq \frac{\Phi_1}{\Phi_2} = 2$, so GR^3 would reset the current group to the largest weight group G_1 and run client C_1. Based on (1), C_1 would be run for three time quanta before selecting G_2 again to run client C_2. After running C_2 the second time, W_2 would increase such that $\frac{W_2+1}{W_3+1} = 3 > \frac{\Phi_1}{\Phi_2} = 2$, so GR^3 would then move on to the last group G_3 and run client C_3. The resulting schedule would then be: G_1, G_1, G_2, G_1, G_1, G_1, G_2, G_3. Each group therefore receives its proportional allocation in accordance with its respective group weight.

The GR^3 **intragroup scheduling** algorithm selects a client from the selected group. All clients within a group have weights within a factor of 2, and all client weights in a group G are normalized with respect to the minimum possible weight, $\phi_{\min} = 2^{\sigma_G}$, for any client in the group. GR^3 then effectively runs each client within a group in round-robin order for a number of time quanta equal to the client's normalized weight, rounded down to the nearest integer value. GR^3 keeps track of fractional time quanta that are not used and accumulates them in a deficit value for each client, then allocates an extra time quantum to a client when its deficit reaches one.

More specifically, the GR^3 intragroup scheduler considers the scheduling of clients in rounds. A *round* is one pass through a group's run queue of clients from beginning to end. The group run queue does not need to be sorted in any manner. During each round, the GR^3 intragroup algorithm considers the clients in round-robin order. For each runnable client C_i, the scheduler determines the maximum number of time quanta that the client can be selected to run in this round as $\lfloor \frac{\phi_i}{\phi_{\min}} + D_{C_i}(r-1) \rfloor$. $D_{C_i}(r)$, the deficit of client C_i after round r, is defined recursively as $D_{C_i}(r) = \frac{\phi_i}{\phi_{\min}} + D_{C_i}(r-1) - \lfloor \frac{\phi_i}{\phi_{\min}} + D_{C_i}(r-1) \rfloor$, with $D_{C_i}(0) = 0$. Thus, in each round, C_i is allotted one

time quantum plus any additional leftover from the previous round, and $D_{C_i}(r)$ keeps track of the amount of service that C_i missed because of rounding down its allocation to whole time quanta. We observe that $0 \leq D_{C_i}(r) < 1$ after any round r so that any client C_i will be allotted one or two time quanta. Note that if a client is allotted two time quanta, it first executes for one time quantum and then execute for the second time quantum the next time the intergroup scheduler selects its respective group again (in general, following a timespan when clients belonging to other groups get to run).

The following example illustrates how GR^3 scheduling works. Consider a set of six clients C_1 through C_6 with weights 12, 3, 3, 2, 2, 2 respectively. The six clients will be put into two groups G_1 and G_2 with respective group order 1 and 3 as follows: $G_1 = \{C_2, C_3, C_4, C_5, C_6\}$ and $G_2 = \{C_1\}$. The weight of the groups are $\Phi_1 = \Phi_2 = 12$. GR^3 intergroup scheduling will consider the groups in this order: G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2, G_1, G_2. G_2 will schedule client C_1 every time G_2 is considered for service since it has only one client. Since $\phi_{\min(G_1)} = 2$, the normalized weights of clients C_2, C_3, C_4, C_5, and C_6 are 1.5, 1.5, 1, 1, and 1, respectively. In the beginning of round 1 in G_1, each client starts with 0 deficit. As a result, the intragroup scheduler will run each client in G_1 for one time quantum during round 1. After the first round, the deficit for C_2, C_3, C_4, C_5, and C_6 are 0.5, 0.5, 0, 0, and 0. In the beginning of round 2, each client gets another $\frac{\phi_i}{\phi_{\min}}$ allocation. As a result, the intragroup scheduler will select clients C_2, C_3, C_4, C_5, and C_6 to run in order for 2, 2, 1, 1, and 1 time quanta, respectively, during round 2. Rounds 3 and 4 are similar. The sequence of clients that the scheduler runs for each unit of time is C_2, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1, C_2, C_1, C_2, C_1, C_3, C_1, C_3, C_1, C_4, C_1, C_5, C_1, C_6, C_1.

2.3 GR^3 Dynamic Considerations

In the previous section, we presented the basic GR^3 scheduling algorithm, but we did not discuss how GR^3 deals with dynamic considerations that are a necessary part of any on-line scheduling algorithm. We now discuss how GR^3 allows clients to be dynamically created, terminated, or change run state.

Clients that are runnable can be selected for execution by the scheduler, while clients that are not runnable cannot. With no loss of generality, we assume that a client is created before it can become runnable, and a client becomes not runnable before it is terminated. As a result, client creation and termination have no effect on the GR^3 run queues.

When a client C_i with weight ϕ_i becomes runnable, it is inserted into group $G(i)$ such that ϕ_i is between $2^{\sigma_{G(i)}}$ and $2^{\sigma_{G(i)}+1} - 1$. If the group was previously empty, the client becomes the current client of the group. If the group was not previously empty, GR^3 inserts the client into the respective group's run queue right before the current client, and will be first serviced after all of the other clients in the group have first been considered for scheduling.

When a newly runnable client C_i is inserted into its respective group $G(i)$, the group needs to be moved to its new position on the ordered group list based on its new group weight. The corresponding group work and group weight need to be updated and the client's deficit needs to be initialized. The group weight is simply incremented by the client's weight. We want to scale the group work of $G(i)$ in a similar manner. Denote $W^{\text{old}}_{G(i)}$ as the group work of $G(i)$ and W^{old}_T as the total work before inserting client C_i, respectively. We then scale the group work $W_{G(i)}$ as follows:

$$W_{G(i)} = \begin{cases} \left\lfloor W^{\text{old}}_T \frac{\phi_i}{\Phi^{\text{old}}_T} \right\rfloor & \text{if } G(i) \text{ was empty} \\ \left\lfloor W^{\text{old}}_{G(i)} \frac{\Phi_{G(i)}}{\Phi^{\text{old}}_{G(i)}} \right\rfloor & \text{otherwise} \end{cases}$$

and update $W_T = W^{\text{old}}_T + W_{G(i)} - W^{\text{old}}_{G(i)}$.

Also, since we have decreased the average work in the group through these operations, we need to set the deficit of C_i so that the future increase in service given to the group because of this decrease should be absorbed by the new client. The goal is to have the impact of a new client insertion be as local as possible, while preserving the relationship among the work of the other clients and groups. We therefore assign an initial deficit as follows:

$$D_{C_i} = \begin{cases} \frac{\phi_i}{\Phi_T} W^{\text{old}}_T - \left\lfloor \frac{\phi_i}{\Phi_T} W^{\text{old}}_T \right\rfloor & \text{if } G(i) \text{ was empty} \\ \frac{\Phi_{G(i)}}{\Phi^{\text{old}}_{G(i)}} W^{\text{old}}_{G(i)} - \left\lfloor \frac{\Phi_{G(i)}}{\Phi^{\text{old}}_{G(i)}} W^{\text{old}}_{G(i)} \right\rfloor & \text{otherwise} \end{cases}$$

Since this deficit is less than 1, the new client is mildly compensated for having to wait an entire round until it gets to run, while not obtaining more credit than other, already runnable, clients.

When a client C_i with weight ϕ_i becomes not runnable, we need to remove it from the group's runqueue. This requires updating the group's weight, which potentially includes moving the group in the ordered group list, as well as adjusting the measure of work received according to the new processor share of the group. This can be achieved in several ways. GR^3 is optimized to efficiently deal with the common situation when a blocked client may rapidly switch back to the runnable state again. This approach is based on "lazy" removal, which minimizes overhead associated with adding and removing a client, while at the same time preserving the service rights and service order of the runnable clients. Since a client blocks when it is running, we know that it will take another full round before the client will be considered again. The only action when a client blocks is to set a flag on the client, marking it for removal. If the client becomes runnable by the next time it is selected, we reset the flag and run the client as usual. Otherwise, we remove the client from $G(i)$. In the latter situation, as in the case of client arrivals, the group may need to be moved to a new position on the ordered group list based on its new group weight. The corresponding group

P	Number of processors.
$\wp^k$	Processor k.
$C(\wp^k)$	Client running on processor k.
F_i	Frontlog for client C_i.

Table 2: GR^3MP Terminology

weight is updated by subtracting the client's weight from the group weight. The corresponding group work is scaled in a similar manner as for client insertion:

$$W_{G(i)} = \left\lceil W^{\text{old}}_{G(i)} \frac{\Phi_{G(i)}}{\Phi^{\text{old}}_{G(i)}} \right\rceil , \tag{2}$$

and the total work counter needs to be updated by the formula $W_T = W^{\text{old}}_T + W_{G(i)} - W^{\text{old}}_{G(i)}$. After having performed these removal operations, we restart the scheduler from the largest weight group in the system.

Whenever a client blocks during round r, we set $D_{C_i}(r) = min(D_{C_i}(r-1) + \frac{\phi_i}{\phi_{\min}} - \lceil W(i,r) \rceil, 1)$, where $W(i,r)$ is the service that the client received during round r until it blocked. This preserves the client's credit in case it returns by the next round, while also limiting the deficit to 1 so that a client cannot gain credit by blocking. However, the group consumes 1 tu (its work is incremented) no matter how long the client runs. Therefore, the client forfeits its extra credit whenever it is unable to consume its allocation.

If the client fails to return by the next round, we may remove it. Having kept the weight of the group to the old value for an extra round has no adverse effects on fairness, despite the slight increase in service seen by the group during the last round. By scaling the work of the group and rounding up, we determine its future allocation and thus make sure the group will not have received undue service. We also immediately restart the scheduler from the first group in the readjusted group list, so that any minor discrepancies caused by rounding may be smoothed out by a first pass through the group list.

3 GR^3 Multiprocessor Scheduler (GR^3MP)

We can extend GR^3 to act as a multi-resource scheduler for a system with P processors scheduling from a single, centralized queue. This simple multiprocessor scheme, which we refer to as GR^3MP, preserves the good fairness and time complexity properties of GR^3 in small-scale multiprocessor systems, which are increasingly common today, even in the form of hyperthreaded processors. Table 2 introduces terminology we use to describe GR^3MP. We first describe the basic GR^3MP scheduling algorithm, then discuss dynamic considerations. To deal with infeasible client weights, we then show how GR^3MP uses its grouping strategy in a novel weight readjustment algorithm that is much more efficient than previous approaches [3].

3.1 Basic GR^3MP Algorithm

GR^3MP uses the same GR^3 data structure, namely an ordered list of groups, each containing clients whose weights are within a factor of 2 from each other. When a processor needs to be scheduled, GR^3MP selects the client that would run next under GR^3, essentially scheduling multiple processors from its central runqueue as GR^3 schedules a single processor. However, there is one obstacle to simply applying a uniprocessor algorithm on a multiprocessor system. Each client can only run on one processor at any given time. As a result, GR^3MP cannot select a client to run that is already running on another processor even if GR^3 would schedule that client in the uniprocessor case. For example, if GR^3 would schedule the same client consecutively, GR^3MP cannot schedule that client consecutively on another processor if it is still running.

To handle this situation while maintaining fairness, GR^3MP introduces the notion of a **frontlog**. The frontlog F_j for some client C_j running on a processor $\wp^k$ ($C_j = C(\wp^k)$) is defined as the number of time quanta for C_j accumulated as C_j gets selected by GR^3 and cannot run because it is already running on $\wp^k$. The frontlog F_j is then queued up on $\wp^k$.

Given a client that would be scheduled by GR^3 but is already running on another processor, GR^3MP uses the frontlog to assign the client a time quantum now but defer the client's use of it until later. Whenever a processor finishes running a client for a time quantum, GR^3MP checks whether the client has a non-zero frontlog, and, if so, continues running the client for another time quantum and decrements its frontlog by one. The frontlog mechanism not only ensures that a client receives its proportional share allocation, it also takes advantage of any cache affinity by continuing to run the client on the same processor.

When a processor finishes running a client for a time quantum and its frontlog is zero, we call the processor 'idle'. GR^3MP schedules a client to run on the idle processor by performing a GR^3 scheduling decision on the central queue. If the selected client is already running on some other processor, we increase its frontlog and repeat the GR^3 scheduling, each time incrementing the frontlog of the selected client, until we find a client that is not currently running. We assign this client to the idle processor for one time quantum. This description assumes that there are least $P+1$ clients in the system. Otherwise, scheduling is easy: each client is simply assigned its own processor.

To illustrate GR^3MP scheduling, suppose we have a dual-processor system and three clients C_1, C_2, and C_3 of weights 3, 2, and 1, respectively. C_1 and C_2 will then be part of the order 1 group (assume C_2 is before C_1 in the round-robin queue of this group), whereas C_3 is part of the order 0 group. The GR^3 schedule is C_2, C_1, C_2, C_1, C_1, C_3. $\wp^1$ will then select C_2 to run, and $\wp^2$ selects C_1. When $\wp^1$ finishes, according to GR^3, it will select C_2 once more,

whereas $\wp^2$ selects C_1. When $\wp^1$ again selects the next GR^3 client, which is C_1, it finds that it is already running on $\wp^2$ and thus we set $F_1 = 1$ and select the next client, which is C_3, to run on $\wp^1$. When $\wp^2$ finishes running C_1 for its second time quantum, it finds $F_1 = 1$, sets $F_1 = 0$ and continues running C_1 without any scheduling decision on the GR^3 queue.

3.2 GR^3MP Dynamic Considerations

GR^3MP basically does the same thing as the GR^3 algorithm under dynamic considerations. However, the frontlogs used in GR^3MP need to be accounted for appropriately. If some processors have long frontlogs for their currently running clients, newly arriving clients may not be run by those processors until their frontlogs are processed, resulting in bad responsiveness for the new clients. Although in between any two client arrivals or departures, some processors must have no frontlog, the set of such processors can be as small as a single processor. In this case, newly arrived clients will end up competing with other clients already in the run queue only for those few processors, until the frontlog on the other processors is exhausted.

GR^3MP provides fair and responsive allocations by creating frontlogs for newly arriving clients. Each new client is assigned a frontlog equal to a fraction of the total current frontlog in the system based on its proportional share. Each processor now maintains a queue of frontlog clients and a new client with a frontlog is immediately assigned to one of the processor frontlog queues. Rather than running its currently running client until it completes its frontlog, each processor now round robins among clients in its frontlog queue. Given that frontlogs are small in practice, round-robin scheduling is used for frontlog clients for its simplicity and fairness. GR^3MP balances the frontlog load on the processors by placing new frontlog clients on the processor with the smallest frontlog summed across all its frontlog clients.

More precisely, whenever a client C_i arrives, and it belongs in group $G(i)$, GR^3MP performs the same group operations as in the single processor GR^3 algorithm. GR^3MP finds the processor $\wp^k$ with the smallest frontlog, then creates a frontlog for client C_i on $\wp^k$ of length $F_i = F_T \frac{\phi_i}{\Phi_T}$, where F_T is the total frontlog on all the processors. Let $C_j = C(\wp^k)$. Then, assuming no further clients arrive, $\wp^k$ will round-robin between C_j and C_i and run C_i for F_i and C_j for F_j time quanta.

When a client becomes not runnable, GR^3MP uses the same lazy removal mechanism used in GR^3. If it is removed from the runqueue and has a frontlog, GR^3MP simply discards it since each client is assigned a frontlog based on the current state of the system when it becomes runnable again.

3.3 GR^3 MP Weight Readjustment

Since no client can run on more than one processor at a time, no client can consume more than a $1/P$ fraction of the processing in a multiprocessor system. A client C_i with weight ϕ_i greater than Φ_T/P is considered *infeasible* since it cannot receive its proportional share allocation ϕ_i/Φ_T. GR^3MP will simply assign such a client its own processor to run on. However, since the scheduler uses client weights to determine which client to run, an infeasible client's weight must be adjusted so that it is feasible to ensure that the scheduling algorithm runs correctly to preserve fairness. GR^3MP potentially needs to perform weight readjustment whenever a client is inserted or removed from the runqueue to make sure that all weights are feasible (i.e., the weight of a client is no larger than $\frac{1}{P}$ of the total weight after weight readjustment is completed).

GR^3MP leverages its grouping strategy to perform efficient weight readjustment. GR^3MP starts with the unmodified client weights and maintains a 'saved' list of groups ordered by group weight based on the unmodified client weights. Given the clients whose weights had been adjusted, we determine the group to which each such client belongs based on its original weight, add the client to that group and restore the group in the ordered list of groups according to its position in the 'saved' list. Once the active GR^3 group list has been restored to be an exact copy of the saved group list, GR^3MP uses the following weight readjustment algorithm to construct the set I of infeasible clients and adjust their weights to be feasible. We denote by $|I|$ the cardinality of I and by Φ_I the sum of weights of the clients in I, $\sum_{C \in I} \phi_C$.

GR^3MP starts with I initially empty ($|I| = \Phi_I = 0$), and then proceed from the group containing the largest weight clients towards the group containing the smallest weight clients. For each such group G, if $|G| < P - |I|$ and $2^{\sigma_G} > \frac{\Phi_T - \Phi_I - \Phi_G}{P - |I| - |G|}$, then all the clients in G are infeasible, so that GR^3MP sets $I = I \cup G$ and continues with the next group. Otherwise, GR^3MP knows that all the clients not in $I \cup G$ are feasible and it only needs to find which, if any, clients from G are infeasible. If $|G| \geq 2(P - |I|)$, GR^3MP can stop searching for infeasible clients since all clients $C \in G$ are feasible: $\phi_C < 2^{\sigma_G + 1} \leq 2\frac{1}{|G|}\Phi_G \leq \frac{1}{P-|I|}\Phi_G \leq \frac{1}{P-|I|}(\Phi_T - \Phi_I)$.

Otherwise, if $|G| < 2(P - |I|)$, GR^3MP needs to search through G to determine which clients are infeasible. If the number of clients in G is small, we can sort all clients in G by weight. Then, starting from the largest weight client in G, identify each client $C \in G$ as infeasible and add to the infeasible set I if $\phi_C > \frac{1}{P-|I|}(\Phi_T - \Phi_I)$. If the inequality does not hold, we are finished since all clients of less weight than C will be feasible as well.

GR^3MP can alternatively use a more complicated but lower time complexity divide-and-conquer algorithm to find the infeasible clients in G. In this case, GR^3MP par-

titions G around its median C into G_S, the set of G clients that have weight less than ϕ_C and G_B, the set of G clients that have weight larger than ϕ_C. If $\phi_C > \frac{\Phi_T - \Phi_I - \Phi_{G_B}}{P - |I| - |G_B|}$, then all clients in G_B are infeasible, and we therefore set $I = I \cup G_B \cup \{C\}$ and recurse on G_S to find all infeasible clients. Otherwise, all clients in G_S are feasible, and thus we recurse on G_B to find all infeasible clients. The algorithm finishes when either $|I| = P$ or the set we need to recurse on is empty.

Once all infeasible clients have been identified, GR^3MP determines the sum of the weights of all feasible clients, $\Phi_T^f = \Phi_T - \Phi_I$. We can now compute the new total share in the system as $\Phi_T = \frac{P}{P-|I|}\Phi_T^f$, namely the solution to the equation $\Phi_T^f + I\frac{x}{P} = x$. Once we have the adjusted Φ_T, we change all the weights for the infeasible clients in I to $\frac{\Phi_T}{P}$.

Given the adjusted client weights, we alter the 'active' GR^3 group structure to reflect the new client weights and the weight ratios among groups. Specifically, we remove the infeasible clients from their respective groups, and put them all in the same group, since their adjusted weights will be equal. Empty groups (the ones that contained only infeasible clients) are then disconnected from the group list.

4 GR^3 Fairness and Complexity

We analyze the fairness and complexity of GR^3 and GR^3MP . To analyze fairness, we use a more formal notion of proportional fairness defined as *service error*, a measure widely used [8, 10, 17, 21] in the analysis of scheduling algorithms. For simplicity, we assume that clients are always runnable in the following analysis.

We use a strict measure of service error relative to Generalized Processor Sharing (GPS) [13], an idealized model that achieves *perfect fairness*: $W_C = W_T\frac{\phi_C}{\Phi_T}$, an ideal state in which each client always receives service exactly proportional to its share. Although all real-world scheduling algorithms must time-multiplex resources in time units of finite size and thus cannot maintain perfect fairness, some algorithms stay closer to perfect fairness than others and therefore have less service error. We quantify how close an algorithm gets to perfect fairness using the *client service time error*, which is effectively the difference between the service received by client C and its share of the total work done by the processor: $E_C = W_C - \phi_C\frac{W_T}{\Phi_T}$. A positive service time error indicates that a client has received more than its ideal share over a time interval; a negative error indicates that it has received less. To be precise, the error E_C measures how much time a client C has received beyond its ideal allocation. The goal of a proportional share scheduler should be to minimize the absolute value of the allocation error between clients with minimal scheduling overhead.

We analyze the fairness of GR^3 and GR^3MP by providing bounds on the service error. To do this, we define two other measures of service error. The *group service time error* is a similar measure for groups that quantifies the fairness of allocating the processor among groups: $E_G = W_G - \Phi_G\frac{W_T}{\Phi_T}$. The *group relative service time error* represents the service time error of client C if there were only a single group G in the scheduler and is a measure of the service error of a client with respect to the work done on behalf of its group: $E_{C,G} = W_C - \phi_C\frac{W_G}{\Phi_G}$. We first show bounds on the group service error of the intergroup scheduling algorithm. We then show bounds on the group relative service error of the intragroup scheduling algorithm. Finally, we combine these results to obtain the client service error bounds for the overall scheduler. We also discuss the scheduling overhead of GR^3 and GR^3MP in terms of their time complexity. We show that both algorithms can make scheduling decisions in $O(1)$ time with $O(1)$ service error given a constant number of groups.

4.1 Analysis of GR^3

Intergroup Fairness To demonstrate the fairness mechanism of GR^3, we begin by assuming the weight ratios of consecutive groups in the group list are integers. For this case, we state and prove the following:

Lemma 1 *If* $\frac{\Phi_j}{\Phi_{j+1}} \in \boldsymbol{N}$, $1 \le j < g$, *then* $-1 < E_{G_k} \le (g-k)\frac{\Phi_k}{\Phi_T}$ *for any group* G_k.

Proof: Let us consider the decision faced by the intergroup scheduler after having selected some group G_j. When W_j becomes $(W_{j+1}+1)\frac{\Phi_j}{\Phi_{j+1}} - 1$, (1) is still false (we have an equality), and so it will take another selection of G_j before GR^3 will move on to G_{j+1}. Therefore, after G_{j+1} is selected, the ratio of the work of the two consecutive groups equals the ratio of their weights:

$$\frac{W_j}{\Phi_j} = \frac{W_{j+1}}{\Phi_{j+1}}. \tag{3}$$

In particular, let the last selected group be G_k; then we know (3) holds for all $1 \le j < k$. If $j > k$, then we know (3) held after G_{j+1} was selected. Until the next time that G_{j+1} gets selected again and (3) holds once more, W_j can only increase, with W_{j+1} fixed. Thus,

$$\frac{W_{j+1}}{\Phi_{j+1}} \le \frac{W_j}{\Phi_j} \le \frac{W_{j+1}+1}{\Phi_{j+1}} - \frac{1}{\Phi_j}. \tag{4}$$

The right inequality in (4) is simply the negation (1), slightly rearranged. For the particular case when $j = k$, we can write based on (4) just before having selected G_k (when W_k was less by 1):

$$\frac{W_{k+1}}{\Phi_{k+1}} + \frac{1}{\Phi_k} \le \frac{W_k}{\Phi_k} \le \frac{W_{k+1}+1}{\Phi_{k+1}}. \tag{5}$$

By summing (4) over $k < j < i$ and adding (5), we get $\frac{W_i}{\Phi_i} + \frac{1}{\Phi_k} \le \frac{W_k}{\Phi_k} \le \frac{W_i+1}{\Phi_i} \forall i, k < i \le g$. Also, from (3), we have $\frac{W_k}{\Phi_k} = \frac{W_i}{\Phi_i} \forall i, 1 \le i < k$.

Multiplying by Φ_i and summing over all i, we get

$$W_T + \frac{1}{\Phi_k}\sum_{i=k+1}^{g}\Phi_i \le W_k\frac{\Phi_T}{\Phi_k} \le W_T + g - k. \quad (6)$$

Therefore, right after G_k is selected, its error $E_{G_k} = W_k - W_T\frac{\Phi_k}{\Phi_T}$ lies between $\frac{1}{\Phi_T}\sum_{i=k+1}^{g}\Phi_i \in (0,1)$ and $(g-k)\frac{\Phi_k}{\Phi_T}$. Since the minimum error will occur right before some selection of G_k, when W_k is less by 1 than in the above analysis, we can bound the negative error by -1. □

We immediately observe that the error is maximized for $k = 1$; thus:

Corollary 1 *If* $\frac{\Phi_j}{\Phi_{j+1}} \in \boldsymbol{N}$, $1 \le j < g$, *then* $-1 < E_G < (g-1)$ *for any group* G.

In the general case, we get similar, but slightly weaker bounds.

Lemma 2 *For any group* G_k, $-\frac{(g-k)(g-k-1)}{2}\frac{\Phi_k}{\Phi_T} - 1 < E_{G_k} < g - 1$.

Proof: The proof follows reasoning similar to that of the previous lemma, starting with some important remarks related to selecting some group G_j.

Let us negate (1) under the form:

$$\frac{W_j + 1}{\Phi_j} \le \frac{W_{j+1} + 1}{\Phi_{j+1}} \quad (7)$$

After having selected G_j, GR^3 will select G_{j+1} if and only if (7) is violated.

First, we make the observation that (7) was invalid just before G_{j+1} was selected, but held the previous time when G_j was selected. Thus,

$$\frac{W_j}{\Phi_j} \le \frac{W_{j+1}}{\Phi_{j+1}} < \frac{W_j + 1}{\Phi_j} \quad (8)$$

holds immediately after G_{j+1} is selected. Furthermore, selecting G_{j+1} has the consequence of making (7) valid again, since

$$\frac{W_j}{\Phi_j} + \frac{1}{\Phi_j} \le \frac{W_{j+1}}{\Phi_{j+1}} + \frac{1}{\Phi_j} \le \frac{W_{j+1}}{\Phi_{j+1}} + \frac{1}{\Phi_{j+1}}. \quad (9)$$

Also, the right inequality in (8) is true in general, since after G_{j+1} was selected and (8) held, W_j could have only increased, while W_{j+1} stayed fixed.

Now assume that the last group to have been selected is G_k. Based on the above, we will derive relationships between W_j and W_{j+1} depending on whether $j < k$, $j = k$, or $j > k$.

1. $j < k$ When G_k is selected, we know that G_{j+1} was selected right after G_j for all $j < k$, and so (8) holds for all $j < k$.
2. $j > k$ We use (9) and the right inequality of (8) to obtain
$$\frac{W_{j+1}}{\Phi_{j+1}} < \frac{W_j + 1}{\Phi_j} \le \frac{W_{j+1} + 1}{\Phi_{j+1}} \quad (10)$$
3. $j = k$ The situation right before G_k got selected is an instance (with $j = k$) of the previous case, with the observation that the new W_k is the old $W_k + 1$. Thus,
$$\frac{W_{k+1}}{\Phi_{k+1}} < \frac{W_k}{\Phi_k} \le \frac{W_{k+1} + 1}{\Phi_{k+1}} \quad (11)$$

Let us now turn our attention to finding the bounds for the group errors.

By summing up (8) over j between i and $k-1$, we get

$$\frac{W_i}{\Phi_i} \le \frac{W_k}{\Phi_k} < \frac{W_i}{\Phi_i} + \sum_{j=i}^{k-1}\frac{1}{\Phi_j},\ 1 \le i < k.$$

Similarly, by summing up (10) over j between $k+1$ and i and adding (11), we get

$$\frac{W_i}{\Phi_i} - \sum_{j=k+1}^{i-1}\frac{1}{\Phi_j} < \frac{W_k}{\Phi_k} \le \frac{W_i}{\Phi_i} + \frac{1}{\Phi_i},\ k < i \le g.$$

We can multiply the above inequalities by Φ_i to obtain:

$$W_i \le W_k\frac{\Phi_i}{\Phi_k} < W_i + \sum_{j=i}^{k-1}\frac{\Phi_i}{\Phi_j},\ 1 \le i < k. \quad (12)$$

and, respectively,

$$W_i - \sum_{j=k+1}^{i-1}\frac{\Phi_i}{\Phi_j} < W_k\frac{\Phi_i}{\Phi_k} \le W_i + 1,\ k < i \le g. \quad (13)$$

Adding (12) over $1 \le i < k$ with (13) over $k < i \le g$, and with the identity $W_k = W_k\frac{\Phi_k}{\Phi_k}$, we get:

$$W_T - \sum_{i=k+1}^{g}\sum_{j=k+1}^{i-1}\frac{\Phi_i}{\Phi_j} < W_k\frac{\Phi_T}{\Phi_k} \le W_T + \sum_{i=1}^{k-1}\sum_{j=i}^{k-1}\frac{\Phi_i}{\Phi_j} + g - k. \quad (14)$$

We notice that $\sum_{i=1}^{k-1}\sum_{j=i}^{k-1}\frac{\Phi_i}{\Phi_j} = \sum_{i=1}^{k-1}(\frac{1}{\Phi_i}\sum_{j=1}^{i}\Phi_j) < \sum_{i=1}^{k-1}(\frac{\Phi_T}{\Phi_i}) < (k-1)\frac{\Phi_T}{\Phi_k}$.
Also, $\sum_{i=k+1}^{g}\sum_{j=k+1}^{i-1}\frac{\Phi_i}{\Phi_j} \le \sum_{i=k+1}^{g}\sum_{j=k+1}^{i-1}1 = \sum_{i=k+1}^{g}(i-k-1) = \frac{(g-k)(g-k-1)}{2}$.
(14) then yields
$W_T - \frac{(g-k)(g-k-1)}{2} < W_k\frac{\Phi_T}{\Phi_k} < W_T + (k-1)\frac{\Phi_T}{\Phi_k} + g - k$,
or
$W_T\frac{\Phi_k}{\Phi_T} - \frac{(g-k)(g-k-1)}{2}\frac{\Phi_k}{\Phi_T} < W_k < W_T\frac{\Phi_k}{\Phi_T} + (k-1) + (g-k)\frac{\Phi_k}{\Phi_T}$, and, since $\frac{\Phi_k}{\Phi_T} \le 1$,

$$W_T\frac{\Phi_k}{\Phi_T} - \frac{(g-k)(g-k-1)}{2} < W_k < W_T\frac{\Phi_k}{\Phi_T} + (g-1).$$

We rewrite the last relation using the definition for the error: $E_{G_k} = W_k - W_T \frac{\Phi_k}{\Phi_T}$ to get

$$-\frac{(g-k)(g-k-1)}{2} < E_{G_k} < (g-1).$$

The above holds right after G_k is selected. To bound E_{G_k} in general, we note that the minimum of E_{G_k} can only occur right before W_k is incremented (group G_k is selected), while the maximum is reached right after the selection of G_k. Hence, subtracting 1 on the negative side concludes the proof. □

It is clear that the lower bound is minimized when setting k=1. Thus, we have

Corollary 2 $-\frac{(g-1)(g-2)}{2}\frac{\phi_G}{\Phi_T} - 1 < E_G < g-1$ *for any group* G.

Intragroup Fairness Within a group, all weights are within a factor of 2 and the group relative error is bound by a small constant.

Lemma 3 $-3 < E_{C,G} < 4$ *for any client* $C \in G$.

Proof: The intragroup Round Robin runs the clients within a group G of order σ_G in the order of the group queue. After r rounds, $W_A = r\frac{\Phi_A}{2^{\sigma_G}} - D_A$ holds for any client A in G. Then, $W_G = \sum_{A\in G} W_A = r\frac{\Phi_G}{2^{\sigma_G}} - \sum_{A\in G} D_A$, and so $E_{C,G} = W_C - W_G\frac{\phi_C}{\Phi_G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C$ for any client $C \in G$ after a round.

Depending on the position of C in the group's queue, the error in general will be different from the error in between rounds.

Worst-case, if C happens to be at the head of G's queue, right after it runs, $E_{C,G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C + \frac{\phi_C}{2^{\sigma_G}} < N_G\frac{\phi_C}{\Phi_G} + \frac{\phi_C}{2^{\sigma_G}} < N_G\frac{2^{\sigma_G+1}}{N_G 2^{\sigma_G}} + \frac{2^{\sigma_G+1}}{2^{\sigma_G}} = 4$.

Similarly, C might be at the tail of the queue, in which case, right before it runs, $E_{C,G} = \frac{\phi_C}{\Phi_G}\sum_{A\in G} D_A - D_C - \frac{\phi_C}{2^{\sigma_G}} > -1 - \frac{\phi_C}{2^{\sigma_G}} > -1 - \frac{2^{\sigma_G+1}}{2^{\sigma_G}} = -3$. □

Overall Fairness of GR^3 Given the error bounds for the inter- and intragroup scheduling algorithms, we can analyze the overall GR^3 fairness.

Lemma 4 *For any client* $C \in G, E_C = E_{C,G} + \frac{\phi_C}{\Phi_G}E_G$.

Proof: $E_C = W_C - W_T\frac{\phi_C}{\Phi_T} = W_C - W_G\frac{\phi_C}{\Phi_G} + W_G\frac{\phi_C}{\Phi_G} - W_T\frac{\phi_C}{\Phi_T} = E_{C,G} + \frac{\phi_C}{\Phi_G}(W_G - W_T\frac{\Phi_G}{\Phi_T})$. □

We are now ready to state the following result to bound the service error relative to GPS of any client in the scheduler to a $O(g)$ positive service error bound and a $O(g^2)$ negative service error bound:

Theorem 1 $-\frac{(g-1)(g-2)}{2}\frac{\phi_C}{\Phi_T} - 4 < E_C < g+3$ *for any client* C.

Proof: Follows from Corollary 2, Lemma 3, and Lemma 4 where we have $\frac{\phi_C}{\Phi_G} \leq 1$. □

In the desirable case when the group weight ratios are integers, we have the tighter bounds of

Theorem 2 *If* $\frac{\Phi_j}{\Phi_{j+1}} \in \boldsymbol{N}$, $1 \leq j < g$ *then* $-4 < E_C < g+3$ *for any client* C.

Proof: Follows from Corollary 1, Lemma 3, and Lemma 4 where we have $\frac{\phi_C}{\Phi_G} \leq 1$. □

Time Complexity of GR^3 GR^3 manages to bound its service error by $O(g^2)$ while maintaining a strict $O(1)$ scheduling overhead. The intergroup scheduler either selects the next group in the list, or reverts to the first one, which takes constant time. The intragroup scheduler is even simpler, as it just picks the next client to run from the unordered round robin list of the group. Adding and removing a client is worst-case $O(g)$ when a group needs to be relocated in the ordered list of groups. This could of course be done in $O(\log g)$ time (using binary search, for example), but the small value of g in practice does not justify a more complicated algorithm.

The *space complexity* of GR^3 is $O(g) + O(N) = O(N)$. The only additional data structure beyond the unordered lists of clients is an ordered list of length g to organize the groups.

4.2 Analysis of GR^3MP

Overall Fairness of GR^3MP Given feasible client weights after weight readjustment, the service error for GR^3MP is bounded below by the GR^3 error, and above by a bound which improves with more processors.

Theorem 3 $-\frac{(g-1)(g-2)}{2}\frac{\phi_C}{\Phi_T} - 4 < E_C < 2g + 10 + \frac{(g-1)(g-2)}{2P}$ *for any client* C.

Proof: Let us denote the total number of scheduling decisions performed on the GR^3 queue by $\overline{W_T} = \sum_{i=1}^{N}\overline{W_i}$, where $\overline{W_i}$ is the number of time quanta that GR^3 would have allocated to client C_i so far. Since a client may not be able to consume all of this allocation and instead frontlog some of it, in general $\overline{W_i} = W_i + F_i$.

Assume first that no client has a frontlog. Then the service received by each client rounded up to integer time quanta is the same as in uniprocessor GR^3 at some time t which is a multiple of the basic time quantum. In particular, when a client C_i just finishes running for a time quantum, $W_i - \phi_i\frac{\sum_{j=1}^{N} W_j}{\Phi_T} \leq W_i - \phi_i\frac{\sum_{j=1}^{N}\lfloor W_j\rfloor}{\Phi_T} \leq (g+3)$. Also, on the negative side, right before C_i starts running, $W_i - \phi_i\frac{\sum_{j=1}^{N} W_j}{\Phi_T} \geq W_i - \phi_i\frac{\sum_{j=1}^{N}\lceil W_j\rceil}{\Phi_T} \geq -\frac{(g-1)(g-2)}{2}\frac{\phi_i}{\Phi_T} - 4$. Since a client has weight no larger than $\frac{\Phi_T}{P}$, it is receiving no less than its due allocation while it is running on a processor. Therefore, the two inequalities above show that

the error in this case is bounded by the error for the single processor case.

Assume now that there are clients with frontlog. Since such clients receive at least their due allocation while they are frontlogged, and since non-frontlogged clients have not skipped any quanta, it is easy to see that the negative error of all clients is bounded by the negative error for the single processor case.

Also, for the frontlogged clients, their allocation is behind their corresponding single processor GR^3 allocation, while for the non-frontlogged clients, their positive error can exceed that for the single processor GR^3 by $\frac{\phi_k}{\Phi_T}\sum F_j$ where the sum is taken over the set of frontlogged clients. Let C_i be a frontlogged client. We would like to bound $F_i^{\max}$, the maximum size of the frontlog on C_i. Writing Theorem 1 for the endpoints of the frontlog inteval (t_1, t_2) where the maximum is reached, and subtracting, we get $\overline{W}(t_2) - \overline{W}(t_1) \leq \frac{\phi_i}{\Phi_T}(t_2 - t_1) + \frac{(g-1)(g-2)}{2}\frac{\phi_i}{\Phi_T} + 4 + (g+3)$. On the other hand, because the client was running continuously on one of the processors, $W_i(t_2) - W_i(t_1) = \frac{1}{P}(t_2 - t_1)$. Since $\frac{\phi_i}{\Phi_T} \leq \frac{1}{P}$, we have $F_i^{\max} = \overline{W}_i(t_2) - \overline{W}_i(t_1) - (W_i(t_2) - W_i(t_1)) \leq \frac{(g-1)(g-2)}{2}\frac{\phi_i}{\Phi_T} + 4 + (g+3)$.

We can thus bound the positive error of non-frontlogged clients C_k by $(g+3)$, the positive error for GR^3, plus $\frac{\phi_k}{\Phi_T}\sum(\frac{(g-1)(g-2)}{2}\frac{\phi_i}{\Phi_T} + 4 + (g+3))$ where the sum is taken over the set of frontlogged clients. Since there are at most $P-1$ frontlogged clients, we can bound the positive error by $(g+3) + \frac{\phi_k}{\Phi_T}(\frac{(g-1)(g-2)}{2} + 4P + P(g+3)) \leq (g+3) + \frac{1}{P}(\frac{(g-1)(g-2)}{2} + P(g+7)) = 2g + 10 + \frac{(g-1)(g-2)}{2P}$. □

Time Complexity of GR^3MP The frontlogs create an additional complication when analyzing the time complexity of GR^3MP . When an idle processor looks for its next client, it runs the simple $O(1)$ GR^3 algorithm to find a client C_i. If C_i is not running on any other processor, we are done, but otherwise we place it on the frontlog and then we must rerun the GR^3 algorithm until we find a client that is not running on any other processor. Since for each such client, we increase its allocation on the processor it runs, the amortized time complexity remains $O(1)$. Nevertheless, we also will bound the time that any single scheduling decision takes. The upper bound is equal to the maximum length of any scheduling sequence of GR^3 consisting of only some fixed subset of $P-1$ clients.

Theorem 4 *The time complexity per scheduling decision in GR^3MP is bounded above by $\frac{(g-k)(g-k+1)}{2} + (k+1)(g-k+1)P$ where $1 \leq k \leq g$.*

Proof: We will use the feasibility constraint on the weight of clients, $\phi_i \leq \frac{\Phi_T}{P}$. We also assume that $N > P$. Otherwise, we are in the trivial case when each client gets its own processor.

Let us now consider a scheduling sequence of GR^3 that consists of a subset S of clients, where $|S| \leq P-1$. Let a_i be the number of clients of group G_i that are also in S ($a_i = |G_i \cap S|$). Clearly, $a_i \leq |G_i|$. Let $k = \min\{i | a_i < |G_i|\}$. That is, k denotes the smallest group not contained in S. Such a k always exists, provided that $N > |S|$, which is true when $N \geq P$.

Let t_1, t_2 be an arbitrary scheduling interval of GR^3. Then, writing (14) at times t_1 and t_2 and subtracting the resulting inequalities, we have (we add a 1 to account for having chosen an arbitrary interval which is not necessarily delimited by instances when G_k has just been selected):

$$\overline{W}_T(t_2 - t_1) \leq \overline{W}_k(t_2 - t_1)\frac{\Phi_T}{\Phi_k} + \sum_{i=k+1}^{g}\sum_{j=k+1}^{i-1}\frac{\Phi_i}{\Phi_j} + 1 + \sum_{i=1}^{k-1}\sum_{j=i}^{k-1}\frac{\Phi_i}{\Phi_j} + g - k.$$

As in the proof of lemma 2, we can bound the two double sums by $\frac{(g-k)(g-k-1)}{2}$ and $(k-1)\frac{\Phi_T}{\Phi_k}$ respectively.

Let us now turn to the first term. Since $a_i = |G_i|$, $\forall i < k$, and no client has weight larger than $\frac{\Phi_T}{P}$, we have $\Phi_i \leq |G_i|\frac{\Phi_T}{P} = a_i\frac{\Phi_T}{P}$. Thus,

$$\sum_{i=1}^{k-1}\Phi_i \leq \sum_{i=1}^{k-1}a_i\frac{\Phi_T}{P} = \frac{\Phi_T}{P}\sum_{i=1}^{k-1}a_i. \tag{15}$$

Therefore,

$$\Phi_T = \sum_{i=1}^{g}\Phi_i \leq \sum_{i=1}^{k-1}\Phi_i + (g-k+1)\Phi_k \leq \frac{\Phi_T}{P}\sum_{i=1}^{k-1}a_i + (g-k+1)\Phi_k.$$

This implies

$$\Phi_k \geq \frac{1}{g-k+1}(\Phi_T - \frac{\Phi_T}{P}\sum_{i=1}^{k-1}a_i). \tag{16}$$

Factoring out Φ_T in (16), we get

$$\frac{\Phi_T}{\Phi_k} \leq \frac{(g-k+1)P}{P - \sum_{i=1}^{k-1}a_i} \tag{17}$$

On the other hand,

$$\sum_{i=1}^{k}a_i \leq \sum_{i=1}^{g}a_i = |S| \leq P-1 \tag{18}$$

hence $P - \sum_{i=1}^{k-1}a_i \geq a_k + 1$.

From here it follows that

$$\frac{\Phi_T}{\Phi_k} \leq \frac{(g-k+1)P}{a_k+1} \tag{19}$$

Therefore, we can bound the first term, $\overline{W}_k(t_2 - t_1)\frac{\Phi_T}{\Phi_k}$, by $\overline{W}_k(t_2 - t_1)\frac{(g-k+1)P}{a_k+1}$.

We now make the observation that $\overline{W}_k(t_2 - t_1) \leq 2a_k$, since the round-robin strategy employed within G_k prevents us from running more than twice each client in $G_k \cap S$ without scheduling clients from $G_k \backslash S$ as well. We can now bound $\overline{W}_T$ as follows:

$$\overline{W}_T \leq 2a_k\frac{(g-k+1)P}{a_k+1} + \frac{(g-k)(g-k-1)}{2} + 1 + (k-1)\frac{\Phi_T}{\Phi_k} + g - k.$$

Using (19), we get:

$$\overline{W}_T \leq 2a_k\frac{(g-k+1)P}{a_k+1} + \frac{(g-k)(g-k-1)}{2} + (k-1)\frac{(g-k+1)P}{a_k+1} + g - k + 1.$$

Since $\frac{a_k}{a_k+1} < 1$ and $a_k + 1 \geq 1$, we simplify this to

$$\overline{W}_T < 2(g-k+1) + \frac{(g-k)(g-k+1)}{2} + (k-1)(g-k+1)P + g - k + 1.$$

Because $\overline{W}_T$ is an integer, we can change the strict inequality to $' \leq'$ by subtracting 1. □

Thus, the length of any schedule consisting of at most $P-1$ clients is $O(g^2P)$. Even when a processor has frontlogs for several clients queued up on it, it will schedule in $O(1)$ time, since it performs round-robin among the frontlogged clients. Client arrivals and departures take $O(g)$ time because of the need to readjust group weights in the saved list of groups. Moreover, if we also need to use the weight readjustment algorithm, we incur an additional $O(P)$ overhead on client arrivals and departures.

Lemma 5 *The complexity of the weight readjustment algorithm is* $O(P)$.

Proof: Restoring the 'saved' group structure will worst case touch a number of groups equal to the number of previously infeasible clients, which is $O(P)$. Identifying the infeasible clients involves iterating over at most P groups in decreasing sequence based on group order, as described in Section 3.3. For the last group considered, we only attempt to partition it into feasible and infeasible clients of its size is less than $2P$. Since partitioning of a set can be done in linear time, and we recurse on a subset half the size, this operation is $O(P)$ as well. □

As a side note, in practice, for small P, the $O(P\log(P))$ sorting approach to determining infeasible clients is simpler and performs better than the $O(P)$ recursive partitioning.

Finally, altering the active group structure to reflect the new weights, is a $O(P+g)$ operation, as two groups may need to be re-inserted in the ordered lis of groups.

5 Measurements and Results

We have implemented GR^3 uniprocessor and multiprocessor schedulers in the Linux operating system and measured its performance. We present some experimental data quantitatively comparing GR^3 performance against other popular scheduling approaches from both industrial practice and research. We have conducted both extensive simulation studies and detailed measurements of real kernel scheduler performance on real applications.

We present simulation results comparing the proportional sharing accuracy of GR^3 and GR^3MP against WRR, WFQ, SFQ, WF^{2}Q, VTRR, and SRR. The simulator enabled us to isolate the impact of the scheduling algorithms themselves and examine the scheduling behavior of these different algorithms across hundreds of thousands of different combinations of clients with different share values. Simulation results are presented in Section 5.1.

We also conducted detailed measurements of real kernel scheduler performance by comparing our prototype GR^3 Linux implementation against the standard Linux scheduler, a WFQ scheduler, and a VTRR scheduler. The experiments we have done quantify the scheduling overhead and proportional share allocation accuracy of these schedulers in a real operating system environment under a number of different workloads. Kernel measurement results are presented in Section 5.2.

All our kernel scheduler measurements were performed on an IBM Netfinity 4500 system with one or two 933 MHz Intel Pentium III CPUs, 512 MB RAM, and 9 GB hard drive. The system was installed with the Debian GNU/Linux distribution version 3.0 and all schedulers were implemented using Linux kernel version 2.4.19. The measurements were done by using a minimally intrusive tracing facility that writes timestamped event identifiers into a memory log and takes advantage of the high-resolution clock cycle counter available with the Intel CPU, providing measurement resolution at the granularity of a few nanoseconds. Getting a timestamp simply involved reading the hardware cycle counter register. We measured the cost of the mechanism on the system to be roughly 35 ns per event.

The kernel scheduler measurements were performed on a fully functional system. All experiments were performed with all system functions running and the system connected to the network. At the same time, an effort was made to eliminate variations in the test environment to make the experiments repeatable.

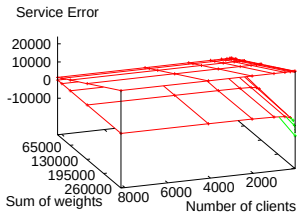

Figure 1: WRR error

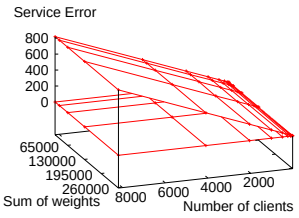

Figure 2: WFQ error

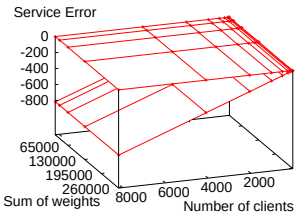

Figure 3: SFQ error

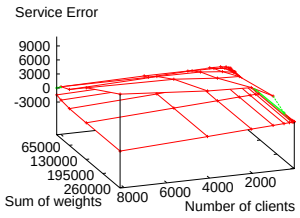

Figure 4: VTRR error

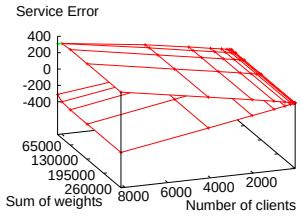

Figure 5: SRR error

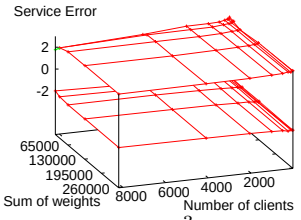

Figure 6: GR^3 error

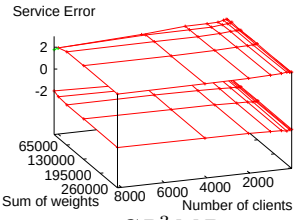

Figure 7: GR^3MP error

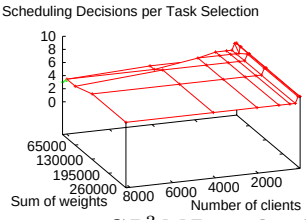

Figure 8: GR^3MP overhead

5.1 Simulation Studies

We built a scheduling simulator that measures the service time error, described in Section 4, of a scheduler on a set of clients. The simulator takes four inputs, the scheduling algorithm, the number of clients N, the total number of shares Φ_T, and the number of client-share combinations. The simulator assigns shares to clients and scales the share values to ensure that they add up to Φ_T. It then schedules the clients using the specified algorithm as a real scheduler would, and tracks the resulting service time error. The simulator runs the scheduler until the resulting schedule repeats, then computes the maximum (most positive) and minimum (most negative) service time error across the nonrepeating portion of the schedule for the given set of clients and share assignments. This process is repeated for the specified number of client-share combinations. We then compute the maximum service time error and minimum service time error for the specified number of client-share combinations to obtain a "worst-case" error range.

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm on 45 different combinations of N and Φ_T (32 up to 8192 clients and 16384 up to 262144 total shares, respectively). Since the proportional sharing accuracy of a scheduler is often most clearly illustrated with skewed weight distributions, one of the clients was given a weight equal to 10 percent of Φ_T. All of the other clients were then randomly assigned shares to sum to the remaining 90 percent of Φ_T. For each pair (N, Φ_T), we ran 2500 client-share combinations and determined the resulting worst-case error ranges.

The worst-case service time error ranges for WRR, WFQ, SFQ, VTRR, SRR, and GR^3 with these skewed share distributions are in Figures 1 to 6. Due to space constraints, WF^{2}Q error is not shown since the results simply verify its known mathematical error bounds of -1 and 1 tu. Each figure consists of a graph of the error range for the respective scheduling algorithm. Each graph shows two surfaces representing the maximum and minimum service time error as a function of N and Φ_T for the same range of values of N and Φ_T. Figure 1 shows WRR's service time error is between -12067 tu and 23593 tu. Figure 2 shows WFQ's service time error is between -1 tu and 819 tu, which is much less than WRR. Figure 3 shows SFQ's service time error is between -819 tu and 1 tu, which is almost a mirror image of WFQ. Figure 4 shows VTRR's service error is between -2129 tu and 10079 tu. Figure 5 shows SRR's service error is between -369 tu and 369 tu.

In comparison, Figure 6 shows the service time error for GR^3 only ranges from -2.5 to 3.0 tu. GR^3 has a smaller error range than all of the other schedulers measured except WF^{2}Q. GR^3 has both a smaller negative and smaller positive service time error than WRR, VTRR, and SRR. While GR^3 has a much smaller positive service error than WFQ, WFQ does have a smaller negative service time error since it is bounded below at -1. Similarly, GR^3 has a much smaller negative service error than SFQ, though SFQ's positive error is less since it is bounded above at 1. Considering the total service error range of each scheduler, GR^3 provides well over two orders of magnitude better proportional sharing accuracy than WRR, WFQ, SFQ, VTRR, and SRR. Unlike the other schedulers, these results show that GR^3 combines the benefits of low service time errors with its ability to schedule in $O(1)$ time.

Note that as the weight skew becomes more accentuated, the service error can grow dramatically. Thus, increasing the skew from 10 to 50 percent results in more than a fivefold increase in the error magnitude for SRR, WFQ, and SFQ, and also significantly worse errors for WRR and VTRR. In contrast, the error of GR^3 is still bounded by small constants: -2.8 and 4.9.

We also measured the service error of GR^3MP us-

ing this simulator configured for an 8 processor system, where the weight distribution was the same as for the uniprocessor simulations above. Note that the client given 0.1 of the total weight was feasible, since $0.1 < \frac{1}{8} = 0.125$. Figure 7 shows GR^3MP's service error is between -2.5 tu and 2.8 tu, slighly better than for the uniprocessor case, a benefit of being able to run multiple clients in parallel. Figure 8 shows the maximum number of scheduling decisions that an idle processor needs to perform until it finds a client that is not running. This did not exceed seven, indicating that the number of decisions needed in practice is well below the worst-case bounds shown in Theorem 4.

5.2 Linux Kernel Measurements

To evaluate the scheduling overhead of GR^3, we compare it against the standard Linux scheduler, a WFQ scheduler, and a VTRR scheduler. We present results from several experiments that quantify how scheduling overhead varies as the number of clients increases. For the first experiment, we measure scheduling overhead for running a set of clients, each of which executed a simple micro-benchmark which performed a few operations in a while loop. A control program was used to fork a specified number of clients. Once all clients were runnable, we measured the execution time of each scheduling operation that occurred during a fixed time duration of 30 seconds. The measurements required two timestamps for each scheduling decision, so measurement error of 70 ns are possible due to measurement overhead. We performed these experiments on the standard Linux scheduler, WFQ, VTRR, and GR^3 for 1 to 400 clients.

Figure 9 shows the average execution time required by each scheduler to select a client to execute on a uniprocessor system and Figure 10 shows the average execution time required by each scheduler to select a client to execute on a dual-processor system. Results for GR^3, VTRR, WFQ, and Linux were obtained on uniprocessor system, and results for GR^3MP and Linux MP were obtained running on a dual-processor system. Dual-processor results for WFQ and VTRR are not shown since MP-ready implementations of them were not available.

For this experiment, the particular implementation details of the WFQ scheduler affect the overhead, so we include results from two different implementations of WFQ. In the first, labeled "WFQ [$O(N)$]", the run queue is implemented as a simple linked list which must be searched on every scheduling decision. The second, labeled "WFQ [$O(\log N)$]", uses a heap-based priority queue with $O(\log N)$ insertion time. To maintain the heap-based priority queue, we used a fixed-length array. If the number of clients ever exceeds the length of the array, a costly array reallocation must be performed. Our initial array size was large enough to contain more than 400 clients, so this additional cost is not reflected in our measurements.

Figure 9 shows the increase in scheduling overhead as the number of clients increases varies a great deal between different schedulers. GR^3 has the smallest scheduling overhead. It requires roughly 300 ns to select a client to execute and the scheduling overhead is essentially constant for all numbers of clients. While VTRR scheduling overhead is also constant, GR^3 has less overhead because its computations are simpler to perform than the virtual time calculations required by VTRR. In contrast, the overhead for Linux and for $O(N)$ WFQ scheduling grows linearly with the number of clients. Both of these schedulers impose more than 200 times more overhead than GR^3 when scheduling a mix of 400 clients. $O(\log N)$ WFQ has much smaller overhead than Linux or $O(N)$ WFQ, but it still imposes significantly more overhead than GR^3, with 8 times more overhead than GR^3 when scheduling a mix of 400 clients. Because of the importance of constant scheduling overhead in server systems, Linux has switched to Ingo Molnar's $O(1)$ scheduler in the recently released Linux 2.6 kernel. However, the Linux 2.6 scheduler shares the poor proportional sharing behavior of Linux 2.4 that we show in the discussion below. Preliminary results also show that GR^3 still runs over 30 percent faster than the Linux 2.6 scheduler on this experiment. Figure 10 shows that GR^3MP provides the same $O(1)$ scheduling overhead on a multiprocessor, although the absolute time to schedule is somewhat higher due to additional costs associated with scheduling in multiprocessor systems. The results show that GR^3MP provides substantially lower overhead than the standard Linux 2.4 scheduler, which suffers from complexity that grows linearly with the number of clients.

As another experiment, we measured the scheduling overhead of the various schedulers for $hackbench$ [18], a benchmark used in the Linux community for measuring scheduler performance with large numbers of processes entering and leaving the run queue at all times. It creates groups of readers and writers, each group having 20 reader tasks and 20 writer tasks, and each writer writes 100 small messages to each of the other 20 readers. This is a total of 2000 messages sent per writer, per group, or 40000 messages per group. We ran a modified version of hackbench to give each reader and each writer a random weight between 1 and 40. We performed these tests on the same set of schedulers for 1 group up to 100 groups. Using 100 groups results in up to 8000 processes running. Because hackbench frequently inserts and removes clients from the run queue, the cost of client insertion and removal is a more significant factor for this benchmark.

Figure 11 shows the average scheduling overhead for each scheduler running on a uniprocessor system and Figure 12 shows the average scheduling overhead for each multiprocessor scheduler running on the dual-processor system. The average overhead is the sum of the times spent on all scheduling events, selecting clients to run and inserting and removing clients from the run queue, divided by

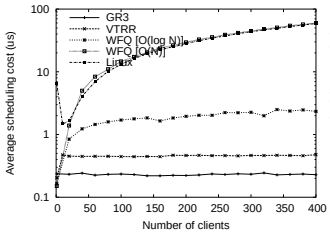

Figure 9: Average scheduling cost (UP)

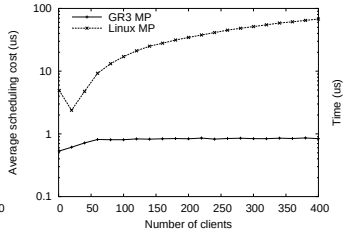

Figure 10: Average scheduling cost (MP)

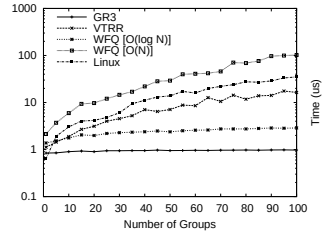

Figure 11: Hackbench scheduling cost (UP)

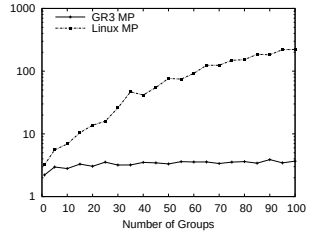

Figure 12: Hackbench scheduling cost (MP)

the number of times the scheduler selected a client to run.

Figure 11 shows the scheduling overhead is higher than the average cost per schedule in Figure 9 for all the schedulers measured since Figure 11 includes a significant component of time due to client insertion and removal from the run queue. The overhead for GR^3 and VTRR remains constant, while the overhead for $O(\log N)$ WFQ, $O(N)$ WFQ and Linux grows with the number of clients. GR^3 still has by far the smallest scheduling overhead among all the schedulers measured. Client insertion, removal, and selection to run in GR^3 are independent of the number of clients. The cost for GR^3 is 3 times higher than before, with client selection to run, insertion, and removal each taking approximately 300 to 400 ns. Figure 12 shows that GR^3MP still has $O(1)$ overhead when running on the multiprocessor system, although the absolute time to schedule is somewhat higher than in the uniprocessor case. The results again show that GR^3MP provides substantially lower overhead than the standard Linux scheduler, whose complexity that grows linearly with the number of clients.

To demonstrate GR^3's efficient proportional sharing of resources on real applications, we briefly describe three simple experiments running web server workloads using the same set of schedulers: GR^3 and GR^3MP Linux uniprocessor and multiprocessor schedulers, WFQ, and VTRR. The web server workload emulates a number of virtual web servers running on a single system. Each virtual server runs the guitar music search engine used at guitarnotes.com, a popular musician resource web site with over 800,000 monthly users. The search engine is a perl script executed from an Apache mod-perl module that searches for guitar music by title or author and returns a list of results. The web server workload configured each server to pre-fork 100 processes, each running consecutive searches simultaneously.

We ran multiple virtual servers with each one having different weights for its processes. In the first experiment, we used six virtual servers, with one server having all its processes assigned weight 10 while all other servers had processes assigned weight 1. In the second experiment, we used five virtual servers and processes assigned to each server had respective weights of 1, 2, 3, 4, and 5. In the third experiment, we ran five virtual servers which assigned a random weight between 1 and 10 to each process. For the Linux scheduler, shares were assigned by selecting `nice` values appropriately. Figures 13 to 18 present the results from the first experiment with one server with weight 10 processes and all other servers with weight 1 processes. The total load on the system for this experiment consisted of 600 processes running simultaneously. For illustration purposes, only one process from each server is shown in the figures. The conclusions drawn from the other experiments are the same, so other results are not shown due to space constraints.

GR^3 and GR^3MP provided the best overall proportional fairness for these experiments while Linux provided the worst overall proportional fairness. Figures 13 to 18 show the amount of processor time allocated to each client over time for the Linux scheduler, WFQ, VTRR, and GR^3. All of the schedulers except GR^3 and GR^3MP have a pronounced "staircase" effect for the search engine process with share 10, indicating that CPU resources are provided in irregular bursts over a short time interval. For the applications which need to provide interactive responsiveness to web users, this can result in extra delays in system response time. The smoother curves for GR^3 and GR^3MP in Figures 16 and 18 show that GR^3 and GR^3MP provide fair resource allocation at a finer granularity than the other schedulers.

6 Related Work

Round robin is one of the oldest, simplest and most widely used proportional share scheduling algorithms. Weighted round-robin (WRR) supports non-uniform client weights by running all clients with the same frequency but adjusting the size of their time quanta in proportion to their respective weights. Deficit round-robin (DRR) [19] was developed to support non-uniform service allocations in packet scheduling. These algorithms have low $O(1)$ complexity but poor short-term fairness, with service errors that can be on the order of the largest client weight in the system. GR^3 uses a novel variant of DRR for intragroup scheduling with $O(1)$ complexity, but also provides $O(1)$ service error by using its grouping mechanism to limit the effective range of client weights considered by the intragroup scheduler.

Fair-share schedulers [7, 11, 12] provide proportional

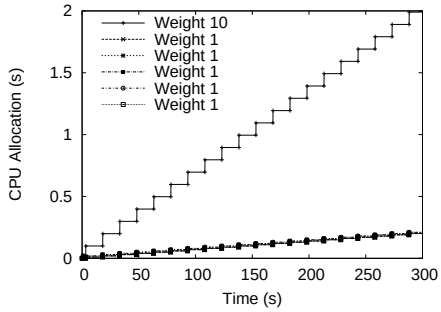

Figure 13: Linux uniprocessor

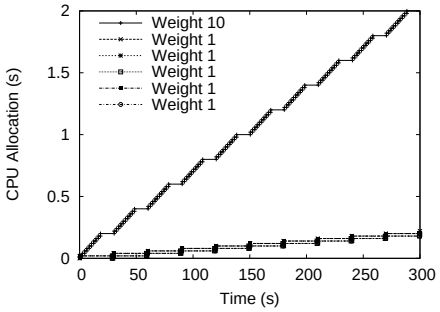

Figure 14: WFQ uniprocessor

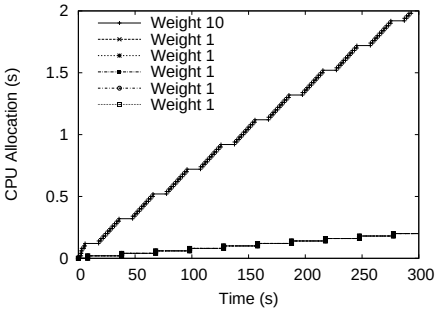

Figure 15: VTRR uniprocessor

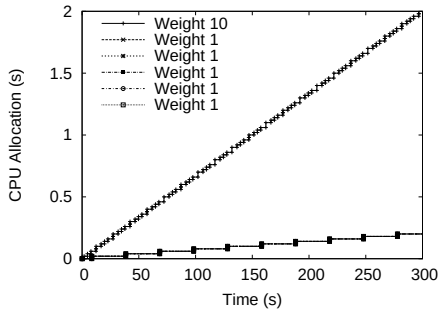

Figure 16: GR^3 uniprocessor

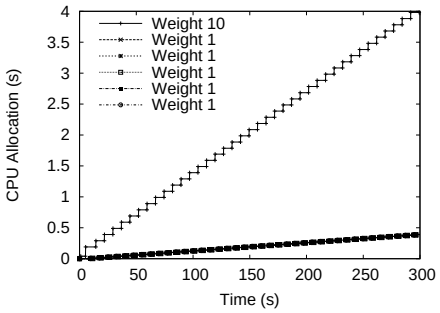

Figure 17: Linux multiprocessor

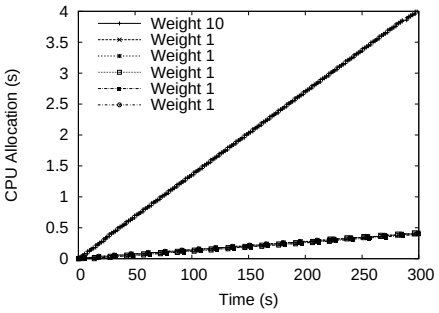

Figure 18: GR^3MP multiprocessor

sharing among users in a way compatible with a UNIX-style time-sharing framework based on multi-level feedback with a set of priority queues. These schedulers typically had low $O(1)$ complexity, but were often ad-hoc and could not provide any proportional fairness guarantees. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals [7].

Lottery scheduling [22] gives each client a number of tickets proportional to its weight, then randomly selects a ticket. Lottery scheduling takes $O(\log N)$ time and relies on the law of large numbers for providing proportional fairness. Thus, its allocation errors can be very large, typically much worse than WRR for clients with smaller weights.

Fair queueing was first proposed by Demers et. al. for network packet scheduling as Weighted Fair Queueing (WFQ) [6], with a more extensive analysis provided by Parekh and Gallager [15], and later applied by Waldspurger and Weihl to CPU scheduling as stride scheduling [22]. Other variants of WFQ such as Virtual-clock [24], SFQ [9], SPFQ [20], and Time-shift FQ [5] have also been proposed. These algorithms generally assign each client a virtual time and schedule the client with the earliest virtual time. These approaches all have $O(\log N)$ time complexity because the clients must be ordered by virtual time. It has been shown that WFQ guarantees that the service time error for any client never falls below -1 [15]. However, WFQ can allow a client to get far ahead of its ideal allocation and accumulate a large positive service time error of $O(N)$, especially with skewed weight distributions.

Several fair queueing approaches have been proposed for reducing this $O(N)$ service time error. A hierarchical scheduling approach reduces service time error to $O(\log N)$. Worst-Case Weighted Fair Queueing [1] introduced eligible virtual times and can guarantee both a lower and upper bound on error of -1 and $+1$, respectively. These algorithms provide stronger proportional fairness guarantees than other approaches, but are more difficult to implement and still require at least $O(\log N)$ time.

Motivated by the need for faster scheduling algorithms with good fairness guarantees [4, 16], novel round-robin scheduling variants such as Virtual-Time Round-Robin (VTRR) [14] and Smoothed Round Robin (SRR) [4] combine the benefits of constant-time scheduling overhead of round-robin with scheduling accuracy that approximates fair queueing. These mechanisms provide proportional sharing by going round-robin through clients in special ways that run clients at different frequencies without having to reorder clients on each schedule. Unlike WRR, they can provide lower service time errors because they do not need to adjust the size of their time quanta to achieve proportional sharing. VTRR combines round-robin scheduling with a virtual time mechanism. In contrast, GR^3's inter-group scheduler relies only on weight ratios and provides better fairness properties even without grouping. SRR introduces a Weight Matrix and Weight Spread Sequence (WSS) and runs tasks simulating a binary counter. Both VTRR and SRR provide proportional sharing with $O(1)$ time complexity for selecting a client to run, though inserting and removing clients from the run queue incur higher overhead: $O(\log N)$ for VTRR and $O(k)$ for SRR , where $k = \log \phi_{\max}$. However, unlike GR^3, both algorithms can suffer from large service time errors especially for skewed weight distributions. For example, we can show that the service error of SRR is worst-case $O(kN)$.

More recently, Stratified Round Robin [16] was pro-

posed as a low complexity solution for network packet scheduling, and possibly CPU scheduling. The algorithm uses a similar grouping strategy as GR^3, distributing all clients with weight between 2^{-k} and $2^{-(k-1)}$ into class F_k. Stratified RR splits time into scheduling slots and then makes sure to assign all the clients in class F_k one slot every scheduling interval, using a credit and deficit scheme within a class. This is also similar to GR^3, with the key difference that a client can run for up to two consecutive time units, while in GR^3, a client is allowed to run only once every time its group is selected regardless of its deficit.

Stratified RR has weaker fairness guarantees and higher scheduling complexity than GR^3. Statified RR assigns each client weight as a fraction of the total processing capacity of the system. This results in weaker fairness guarantees when the sum of these fractions is not close to the limit of 1. For example, if we have $N = 2^k + 1$ clients, one of weight 0.5 and the rest of weight $2^{-(k+2)}$ (total weight = 0.75), Stratified RR will run the clients in such a way that after 2^{k+1} slots, the error of the large client is $\frac{-N}{3}$, such that this client will then run uninterrupted for N tu to regain its due service. Client weights could be scaled to reduce this error, but with additional $O(N)$ complexity. Stratified RR requires $O(g)$ worst-case time to determine the next class that should be selected, where g is the number of groups. Although hardware support can hide this complexity assuming a reasonably small, predefined maximum number of groups [16], running Stratified RR as a CPU scheduler in software still requires $O(g)$ complexity.

GR^3 also differs from Stratified RR and other deficit round-robin variants in its distribution of deficit. In algorithms such as DRR, SRR, Stratified RR, the variation in the deficit of all the clients affects the fairness in the system. To illustrate this, consider $N + 1$ clients, all having the same weight except the first one, whose weight is N times larger. If the deficit of all the clients except the first one is close to 1, the error of the first client will be about $\frac{N}{2} = O(N)$. Therefore, the deficit mechanism itself as employed in round-robin schemes doesn't allow for better than $O(N)$ error. In contrast, GR^3 ensures sure that a group consumes all the work assigned to it, so that the deficit is a tool used only in distributing work within a certain group, and not within the system. Thus, groups effectively isolate the impact of unfortunate distributions of deficit in the scheduler. This allows for the error bounds in GR^3 to depend only on the number of groups instead of the much larger number of clients.

A rigorous analysis on network packet scheduling [23] suggests that $O(N)$ delay bounds are unavoidable with packet scheduling algorithms of less than $O(\log N)$ time complexity. GR^3's $O(g^2)$ error bound and $O(1)$ time complexity are consistent with this analysis, since delay and service error are not equivalent concepts. Thus, if adapted to packet scheduling, GR^3 would worst-case incur $O(N)$ delay while preserving an $O(g^2)$ service error.

For multiprocessor scheduling, Surplus Fair Scheduling (SFS) [3] also adapts a uniprocessor algorithm, SFQ [9], to multiple processors. The authors demonstrate good properties of SFS in practice, but no theoretical fairness bounds are provided. If a selected task is already running on another processor, it is removed from the runqueue. This operation may be expensive and may also introduce unfairness, in particular for low overhead, round-robin type algorithms. In contrast, GR^3MP provides strong fairness bounds with lower scheduling overhead.

SFS introduced the notion of *feasible* tasks along with a $O(P)$-time weight readjustment algorithm, which requires however that the tasks be sorted by their original weight. By using its grouping strategy, GR^3MP performs the same weight readjustment in $O(P)$ time without the need to order clients, thus avoiding the $O(\log N)$ overhead per maintenance operation. The optimality of SFS's and our weight readjustment algorithms rests in preservation of ordering of tasks by weight and of weight proportions among feasible tasks, and not in minimal overall weight change, as [3] claims.

7 Conclusions

We have designed, implemented, and evaluated Group Ratio Round-Robin scheduling in the Linux operating system. We prove that GR^3 is the first and only uniprocessor and multiprocessor scheduling algorithm that simultaneously guarantees $O(1)$ overhead and service error bound of less than $O(N)$ when compared to an idealized processor sharing model. GR^3 achieves these benefits due to its grouping strategy, ratio-based intergroup scheduling, and highly efficient intragroup round robin scheme with good fairness bounds. We have also shown how to adapt GR^3 for a small-scale multiprocessor system while preserving the good bounds on fairness and time complexity. Our experiences with GR^3 show that it is simple to implement and easy to integrate into existing commercial operating systems. We have measured the performance of GR^3 using both simulations and kernel measurements of real system performance using a prototype Linux implementation. Our simulation results show that GR^3 can provide more than two orders of magnitude better proportional fairness behavior than other popular proportional share scheduling algorithms, including WRR, WFQ, SFQ, VTRR, and SRR. Our experimental results using our GR^3 Linux implementation further demonstrate that GR^3 provides accurate proportional fairness behavior on real applications with much lower scheduling overhead than other Linux schedulers, especially for larger workloads.

References

[1] J. Bennett and H. Zhang, "WF^2Q: Worst-case Fair Weighted Fair Queueing," in *Proceedings of INFOCOM '96*, San Francisco, CA, Mar. 1996.

[2] R. Bryant and B. Hartner, "Java technology, threads, and scheduling in Linux" *IBM developerWorks Library Paper*, IBM Linux Technology Center, Jan. 2000.

[3] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors," in *Proceedings of the Fourth ACM Symposium on Operating System Design and Implementation*, Oct. 2000, pp. 45-58.

[4] G. Chuanxiong, "SRR: An 0(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks," in *Proc. of ACM SIGCOMM '01*, Aug. 2001, pp. 211–222.

[5] J. Cobb, M. Gouda, and A. El-Nahas, "Time-Shift Scheduling - Fair Scheduling of Flows in High-Speed Networks," in *IEE/ACM Transactions on Networking*, 1998, pp. 274-285.

[6] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proceedings of ACM SIGCOMM '89*, Austin, TX, Sept. 1989, pp. 1–12.

[7] R. Essick, "An Event-Based Fair Share Scheduler," in *Proceedings of the Winter 1990 USENIX Conference*, USENIX, Berkeley, CA, USA, Jan. 1990, pp. 147–162.

[8] E. Gafni and D. Bertsekas, "Dynamic Control of Session Input Rates in Communication Networks," in *IEEE Transactions on Automatic Control*, 29(10), 1984, pp. 1009–1016.

[9] P. Goyal, H. Vin, and H. Cheng, "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," in *IEEE/ACM Transactions on Networking*, Oct. 1997, pp. 690–704.

[10] E. Hahne and R. Gallager, "Round Robin Scheduling for Fair Flow Control in Data Communication Networks," Tech. Rep. LIDS-TH-1631, LIDS, MIT, Dec. 1986.

[11] G. Henry, "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, 63(8), Oct. 1984, pp. 1845–1857.

[12] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, 31(1), Jan. 1988, pp. 44–55.

[13] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*. New York: John Wiley & Sons, 1976.

[14] J. Nieh, C. Vaill, H. Zhong, "Virtual-time round-robin: An O(1) proportional share scheduler," in *Proceedings of the 2001 USENIX Annual Technical Conference*, USENIX, Berkeley, CA, June 25–30 2001, pp. 245–259

[15] A. Parekh and R. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Transactions on Networking*, 1(3), June 1993, pp. 344–357.

[16] J. Pasquale and S. Ramabhadran, "Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay," *Proceedings of ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003.

[17] K. Ramakrishnan, D. Chiu, and R. Jain, "Congestion Avoidance in Computer Networks with a Connectionless Network Layer, Part IV: A Selective Binary Feedback Scheme for General Topologies," TR. DEC-TR-510, DEC, Nov. 1987.

[18] Hackbench: A New Multiqueue Scheduler Benchmark. `http://www.lkml.org/archive/2001/12/11/19/index.html` Message to Linux Kernel Mailing List, December 2001.

[19] M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round-Robin," in *Proceedings of ACM SIGCOMM '95*, 4(3), Sept. 1995, pp. 231–242.

[20] D. Stiliadis, and A. Varma, "Efficient Fair Queueing Algorithms for Packet-Switched Networks," in *IEEE/ACM Transactions on Networking*, Apr. 1998, pp. 175–185.

[21] R. Tijdeman, "The Chairman Assignment Problem," *Discrete Mathematics*, 32, 1980, pp. 323–330.

[22] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[23] J. Xu and R. Lipton, "On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms," in *Proceedings of ACM SIGCOMM '02*, Pittsburgh, PA, August 2002.

[24] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks," in *ACM Transactions on Computer Systems*, 9(2), May 1991, pp. 101–125.